Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

ANALISI SULLA SICUREZZA DI CONTAINER E IMMAGINI IN AMBIENTE DOCKER

Relatore:
Chiar.mo Prof.
DAVIDE BERARDI

Presentata da:
GABRIELE CRESTANELLO

II Sessione - secondo appello
Anno Accademico 2022/2023

La tecnica senza la fantasia è vuota,
la fantasia senza la tecnica è cieca.

Indice

Introduzione

Nel corso dell'ultima decade il mondo delle tecnologie di virtualizzazione si è reso protagonista di una rapida espansione. Questa tesi ha lo scopo di analizzare gli aspetti riguardanti la sicurezza di questo settore in continua evoluzione, nello specifico in merito alla virtualizzazione basata su container tramite l'utilizzo di Docker.

Il focus iniziale riguarderà l'analisi dei due sistemi per la creazione di ambienti virtuali più diffusi, tramite *hypervisor* o tramite *container* (nello specifico attraverso l'utilizzo di Docker). Successivamente verranno analizzate le feature di sicurezza adottate dall'ecosistema Docker per garantire un compromesso tra protezione e portabilità adeguato, le principali metodologie di attacco a questo tipo di sistemi e alcune contromisure adottate per minimizzarne l'impatto su un sistema multi-tenant, le vulnerabilità e le falle di sicurezza di dominio pubblico reperibili attraverso il dizionario *CVE*, l'applicazione di alcuni degli attacchi presentati in precedenza e le loro tecniche di prevenzione. Per concludere, verrà analizzato il più grande registro di immagini Docker pubbliche, Docker Hub, dal punto di vista delle vulnerabilità presenti.

Capitolo 1

Sistemi di virtualizzazione

Con il termine virtualizzazione si intende il partizionamento di un sistema in ambienti virtuali multipli e isolati, in grado di fornire servizi indipendenti. Riguardo questo processo esistono due approcci principali[2]:

- basato su *container*
- basato su *hypervisor*

In generale, le macchine virtuali consistono in un'astrazione dell'hardware fisico della macchina host, permettendo di trasformare un unico server fisico in molti sistemi isolati e indipendenti tra loro[8].

1.1 Virtualizzazione basata su container

La virtualizzazione tramite container garantisce un approccio più leggero rispetto all'utilizzo di macchine virtuali: la caratteristica principale è quella di operare a livello del sistema operativo host, sfruttandone il kernel per eseguire ambienti virtuali multipli, denominati container[2].

Grazie alla capacità di poter essere integrati in un sistema con istanze multiple, i container garantiscono una migliore condivisione delle risorse e un utilizzo medio dell'hardware più elevato rispetto ad altre implementazioni[3]. È inoltre presente una riduzione dell'esecuzione delle syscall grazie all'utilizzo diretto del kernel della macchina ospite, migliorando ulteriormente le performance rispetto all'implementazione

con hypervisor; questi fattori garantiscono inoltre una riduzione delle dimensioni delle immagini utilizzate[3].

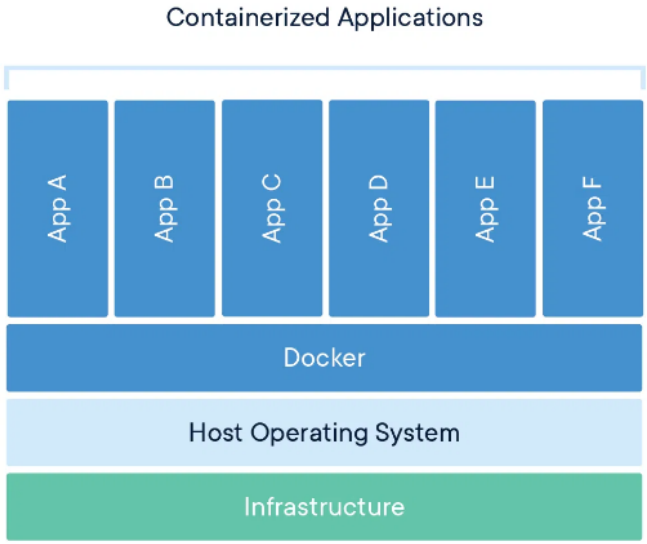

Figura 1.1: Struttura di un sistema con virtualizzazione basata su container[8]

Grazie alla condivisione di un unico kernel l'approccio basato su container è sensibilmente più performante[3] rispetto alla gestione di macchine virtuali con hypervisor, ma risente di alcune vulnerabilità riguardanti la sicurezza derivate dalla possibilità di interfacciamento diretto con il kernel[18].

1.1.1 Introduzione a Docker

Docker è un software libero, progettato per lo sviluppo, la distribuzione e l'esecuzione di applicazioni in ambienti isolati, chiamati *container*[6]: a livello di piattaforma, si è reso protagonista di una rapida crescita dal punto di vista dell'adozione, al punto da rientrare negli strumenti di sviluppo utilizzati per progetti del calibro di Ebay e Spotify[2].

La particolarità di questo strumento risiede nella possibilità di offrire un servizio agnostico rispetto a hardware e piattaforma sottostanti, rendendo il processo di sviluppo indipendente da linguaggi, framework e tecnologie utilizzate[18].

A livello di architettura può essere ridotto ad un sistema client-server, delegando al Docker daemon (*dockerd*, presente nel sistema host) le task legate a costruzione, esecuzione e distribuzione dei container. La comunicazione tra client e daemon è

implementata attraverso un'API REST su socket UNIX o su un'interfaccia di rete[6]. Per l'esecuzione di applicazioni Docker si avvale di due tipologie di oggetti:

- le *immagini*, dei template in sola lettura contenenti le istruzioni per la creazione dei container[6]
- i *container*, le singole entità virtuali e isolate, costituite da un'istanza eseguibile di un'immagine[6]

Immagini

Le immagini utilizzate da Docker consistono in template di sola lettura contenenti le istruzioni per la realizzazione dei container. È possibile utilizzare immagini pubbliche (attraverso l'uso di *registri*) o crearne di nuove, utilizzando un'altra immagine come base di partenza. La creazione di nuove immagini avviene attraverso un *Dockerfile* in cui vengono definiti i passi necessari alla creazione[6].

Container

I container rappresentano le singole istanze di un'immagine eseguibili in maniera isolata. Le operazioni principali consistono in avvio, pausa e rimozione, gestite attraverso l'API di Docker o la sua CLI. È inoltre possibile controllare il livello di isolamento (o connessione) di un container, ad esempio relativamente a storage e rete[6].

Registri

Per la conservazione e condivisione delle immagini il sistema implementato da Docker si avvale dei *registri*, dei repository pubblici (o privati) consultabili liberamente e a cui fare riferimento per le operazioni di *pull* delle immagini[6]; un registro pubblico di immagini di largo utilizzo è *Docker Hub*. È inoltre possibile compiere un'operazione definita di *push*, per l'invio dell'immagine locale al registro configurato.

runC

L'utilità dell'ecosistema Docker risiede nella sua capacità di eseguire applicazioni distribuite su un'ampia varietà di sistemi hardware e configurazioni del sistema operativo sottostante: per rendere tutto ciò possibile è necessario un ambiente dedicato per garantire un'astrazione del sistema in uso senza introdurre eccessivi svantaggi dal punto di vista delle performance[7].

runC è un componente di basso livello per integrare un insieme di feature essenziali per l'esecuzione di ambienti virtuali, che include tutto il codice usato da Docker per interagire con le feature di sistema relative ai container con lo scopo di garantirne la sicurezza, renderlo utilizzabile su larga scala e senza essere dipendente dal resto della piattaforma Docker[7]. Alcune delle sue feature principali riguardano il supporto ai *namespaces*, SELinux, AppArmor, *seccomp* e in generale tutte le feature di sicurezza di Linux, oltre alla possibilità di semplificare il processo di portabilità e migrazione dei container.

1.2 Virtualizzazione basata su hypervisor

La virtualizzazione basata su hypervisor prevede la realizzazione di macchine virtuali complete sopra il livello del sistema operativo host[2].

Questo sistema garantisce una maggiore sicurezza rispetto all'approccio basato su container[18], ma a causa della ridondanza nell'utilizzo dei kernel introduce svantaggi dal punto di vista di spazio e risorse utilizzate, nel numero di ambienti virtuali costruiti e in generale offre peggiori performance rispetto al sistema presentato precedentemente[2].

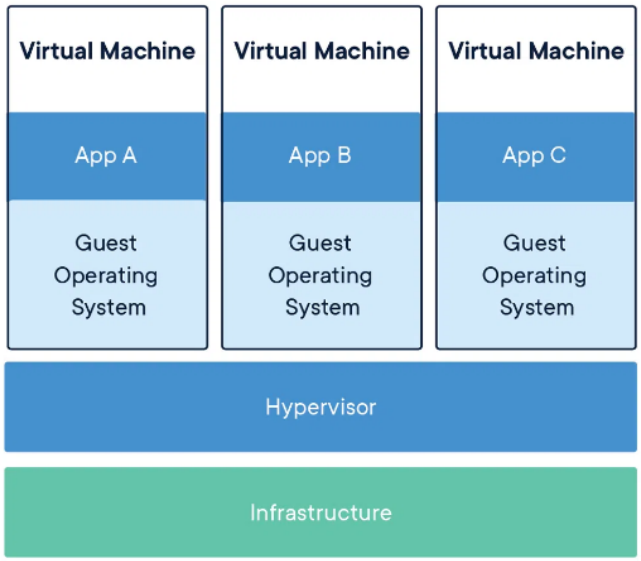

Figura 1.2: Struttura di un sistema con virtualizzazione basata su hypervisor[8]

Per hypervisor si intende un software destinato all'hosting di diverse macchine

virtuali all'interno dello stesso sistema, condividendone le risorse[16]. È possibile individuare 2 tipologie di hypervisor:

- type 1 o *bare-metal hypervisor*, che opera come un sistema leggero sull'hardware sottostante; questa tipologia garantisce una maggiore sicurezza e generalmente migliori prestazioni rispetto alla seguente[16]
- type 2 o *hosted hypervisor*, implementato sopra il livello del sistema operativo host, al pari degli altri processi in esecuzione[16]

L'adozione di entrambe le tipologie fornisce un buon livello di sicurezza e rende possibile l'esecuzione di server virtuali multipli e isolati all'interno di un sistema multi-tenant.

Il tipo più diffuso e il bare-metal che permette l'installazione del software di virtualizzazione direttamente sopra il livello hardware e offre un buon livello di isolamento dal sistema operativo (il quale può essere considerato il target degli attacchi), garantendo così una prevenzione relativa all'escalation di un attacco su più sistemi a partire da un ambiente virtuale compromesso.

Capitolo 2

Analisi sulla sicurezza di Docker

Questo capitolo ha lo scopo di analizzare la sicurezza in ambiente Docker, ponendo un'enfasi sulle feature adottate per garantirne la salvaguardia e i modelli di minaccia in grado di porre un sistema in una situazione di pericolo.

2.1 Feature di sicurezza

L'esecuzione di servizi in un sistema virtuale multi-tenant comporta dei rischi non trascurabili riguardo la sicurezza e la privacy[18]. A tal proposito, l'implementazione basata su hypervisor è ritenuta più efficace in ambito di prevenzione e protezione, a causa del livello di sicurezza aggiuntivo presente tra la macchina virtuale e il kernel host: un attaccante deve essere in grado di bypassare il kernel della macchina virtuale e l'hypervisor, prima di riuscire a comunicare con il kernel del sistema sotto attacco[2].

Al contrario, in un sistema implementato tramite container è possibile comunicare direttamente con il kernel del sistema host: per questo motivo si è resa necessaria l'adozione di sistemi di sicurezza aggiuntivi in questa tipologia di implementazioni[2]. Le prime due feature riguardanti la sicurezza prevedono l'adozione di *namespaces* e *cgroups* dal sistema Linux: i *cgroups* forniscono un meccanismo per la limitazione delle risorse destinate ai processi legati a ciascun container, i *namespaces* permettono di isolare ciascun container[2].

La provenienza degli attacchi contro un sistema può essere esterna, mirata ad ottenere l'accesso all'ambiente virtuale, oppure interna, da parte di un utente malevolo

già presente e con lo scopo di ottenere più privilegi. In entrambe le situazioni lo scopo principale di un attacco è volto al compromettere risorse e dati presenti nel sistema[18].

Le feature di sicurezza trattate a seguito riguardano la protezione da attacchi la cui origine è classificata come interna.

2.1.1 Sistemi di sicurezza implementati da Docker

A seguito verranno analizzati i sistemi di protezione implementati e introdotti dall'adozione di Docker. Nello specifico un sistema con virtualizzazione basata su container dovrebbe disporre dei seguenti livelli di isolamento[14] per prevenire che un container compromesso possa interferire con gli altri container in esecuzione: isolamento dei processi, del filesystem, dei dispositivi, dell'IPC, di rete e prevedere inoltre la possibilità di limitare le risorse a disposizione.

Isolamento dei processi

Lo scopo dell'isolamento dei processi è focalizzato sulla prevenzione dell'utilizzo di interfacce di process management da parte di container compromessi per influenzare o attaccare altri container o sistemi esterni. Questo isolamento è raggiunto racchiudendo i container in esecuzione all'interno di *namespaces*, in modo da limitarne i permessi e la visibilità[2].

Per attuare questo tipo di prevenzione, è necessario un supporto da parte di un sistema di isolamento dei PID (Process ID, identificativo di un processo in esecuzione), in modo che ogni spazio degli ID interno al container sia isolato rispetto a quello del sistema host. I namespaces relativi ai PID prevedono l'utilizzo di una struttura gerarchica, motivo per cui ogni nuovo namespace creato e assegnato ad un container può influenzare solamente i PID "figli", senza poter interagire con i processi in esecuzione in ambienti esterni. A causa di questo livello di isolamento viene reso molto più complesso ogni tentativo di attacco nei cofronti di processi esterni al container.

Isolamento del filesystem

Un ulteriore livello di protezione viene aggiunto grazie al meccanismo di isolamento del filesystem, focalizzato sulla prevenzione dell'accesso e la protezione da modifiche illegittime sui dati memorizzati nel sistema host.

Docker utilizza i *filesystem namespaces* (anche chiamati *mount namespaces*) per isolare la gerarchia del filesystem associata a container diversi con lo scopo di rendere le

modifiche effettuate impattanti solo all'interno del container stesso. Fanno eccezione alcuni percorsi che non rientrano nel namespace fornito:

- */sys*
- */proc/sys*
- */proc/sysrq-trigger*
- */proc/irq*
- */proc/bus*

Per questo motivo un container docker necessita di un'operazione di *mount* prima di poter operare su di essi[2].

La metodologia di prevenzione e protezione di Docker è messa in atto tramite due meccanismi:

1. vengono disabilitati i permessi in scrittura relativi ai container
2. non viene permesso ai processi interni al container di eseguire un'operazione di remount interna al container stesso. Questo sistema di protezione è reso possibile attraverso la rimozione della *capability* *CAP_SYS_ADMIN*[2]

Questo meccanismo messo in atto da parte di Docker limita il rischio che un container compromesso possa accedere e interagire con elementi del filesystem in modo malevolo, prevenendo attacchi di origine interna.

Isolamento dei dispositivi

Nei sistemi Unix il kernel e le applicazioni hanno la possibilità di interagire con le componenti hardware attraverso dei file speciali, chiamati *device nodes* o nodi di dispositivo. Questi file si comportano come vere e proprie interfacce in modo da fare da tramite con i driver dei dispositivi verso cui effettuare l'accesso: alcuni esempi riguardano

- */dev/mem* per l'accesso alla memoria fisica
- */dev/sd** per l'accesso allo storage
- */dev/tty* per l'accesso al terminale

La possibilità che un processo acceda a questi nodi può costituire un grande problema relativo alla sicurezza di tutto il sistema host, motivo per cui vengono predisposte delle limitazioni ai nodi a cui ogni container può accedere[2].

Queste limitazioni riguardo gli accessi sono messe in atto dal *Device Whitelist Controller*, una feature messa a disposizione dall'implementazione dei *cgroups* attraverso la quale Docker

- limita i dispositivi a cui ogni container può accedere
- blocca la creazione di nuovi nodi da parte dei processi interni ai container

La politica di default di Docker prevede di eseguire ogni container con privilegi minimi relativi all'accesso ai dispositivi. Nel caso in cui l'esecuzione di un container avvenga in modalità "privileged", sarebbe garantito l'accesso da parte di quest'ultimo a tutti i nodi, e di conseguenza a tutti i dispositivi connessi.

Isolamento IPC

Con il termine IPC (inter-process communication) si intende un insieme di oggetti utili allo scambio di dati tra processi, in grado di consentirne la comunicazione: alcuni esempi riguardano l'implementazione di segmenti di memoria condivisa, semafori e code di messaggi[2].

Il livello di isolamento relativo alla comunicazione tra processi messo in atto da Docker ha lo scopo di limitare le interferenze container-container e container-host, restringendo l'accesso solamente ad alcune tipologie di risorse IPC. Questa limitazione è messa in atto attraverso l'utilizzo degli *IPC namespaces*[2]. I processi interni ad un namespace possono accedere solamente alle risorse destinate loro in precedenza: Docker assegna un IPC namespace ad ogni container, in modo da prevenire situazioni in cui i processi interni ad un container possano interferire con risorse esterne o del sistema host.

Isolamento di rete

Un ulteriore avanzamento dal punto di vista dell'isolamento per la protezione da attacchi interni riguarda l'isolamento relativo alle risorse di rete. Questo tipo di protezione è di fondamentale importanza per la prevenzione di attacchi di rete, come ad esempio gli attacchi Man-in-the-Middle (MITM) o quelli relativi all'ARP spoofing. Questa tipologia di prevenzione è messa in atto limitando o rimuovendo la possibilità dei container di monitorare o manipolare il traffico di rete in transito degli altri container o del sistema host.

Risulta ricorrente l'utilizzo dei namespaces, in questo caso specifico Docker crea per ogni container un *network namespace* indipendente: ad ogni container vengono

assegnati, tra le varie cose, un indirizzo IP, delle tabelle di routing e dei driver di rete dedicati[2].

La connettività container-container e container-host è permessa attraverso l'utilizzo del *Virtual Ethernet bridge*: questo sistema consente la creazione di un bridge virtuale interno al sistema host con lo scopo di inoltrare i pacchetti tra le interfacce di rete a disposizione[2]. L'adozione di questo sistema, tuttavia, rende i container vulnerabili ad attacchi di tipo MAC flooding e ARP spoofing, a causa dell'inoltro di tutti i pacchetti senza la possibilità di applicare un filtro.

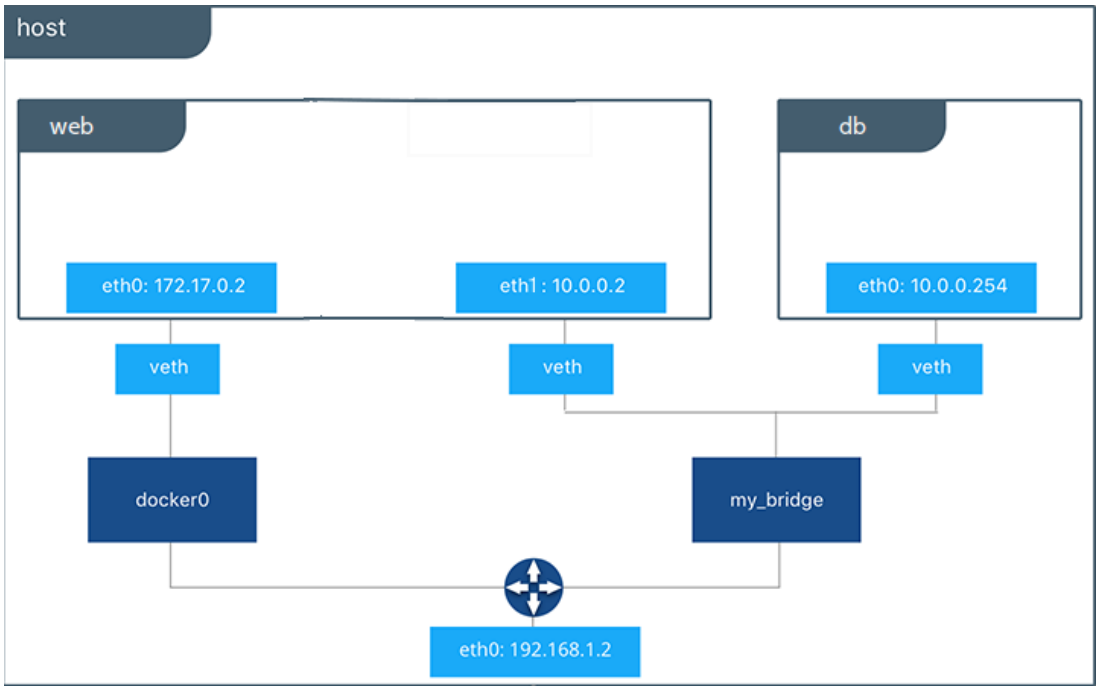

Figura 2.1: Esempio di implementazione di Virtual Ethernet [fonte: docs.docker.com]

Limitazione delle risorse

La prevenzione di attacchi di tipo *Denial-of-Service* (DoS) risulta fondamentale anche per quanto riguarda le tecnologie di virtualizzazione: lo scenario tipico è rappresentato da un sistema multi-tenant in cui uno dei processi in esecuzione tenta di monopolizzare e consumare le risorse a disposizione, causando rallentamenti e interruzioni di servizio per gli altri processi attivi.

L'approccio adottato da Docker prevede la limitazione delle risorse allocate a ciascun container attraverso l'uso dei *cgroup*[2]: è prevista la limitazione di risorse condivise come:

- accesso alla CPU
- memoria centrale
- memorie di massa
- periferiche di input/output

Per ogni container è possibile assegnare una quota di risorse utilizzabile, in modo da prevenire abusi da parte dei processi in esecuzione.

2.1.2 Sistemi di sicurezza offerti dal kernel Linux

Docker, essendo un metodo di virtualizzazione eseguito a livello del sistema operativo host, è più vulnerabile ad attacchi che prevedono l'interazione con il kernel rispetto alla virtualizzazione basata su hypervisor. Vengono adottate alcune strategie messe a disposizione dal kernel Linux, analizzate a seguito, con lo scopo di limitare le interazioni tra i container e l'infrastruttura sottostante.

Capabilities

Tradizionalmente i sistemi UNIX suddividono i tipi di processi in due categorie[9], basandosi sui permessi concessi:

- processi *privileged* (il cui proprietario è superuser/root)
- processi *unprivileged* (il cui proprietario è un normale utente)

Ai processi di tipo privileged è concesso di bypassare tutti i controlli relativi ai permessi messi in atto dal kernel del sistema, mentre quelli unprivileged sono regolarmente sottoposti a controlli basati sulle credenziali del processo[9].

A partire dalla versione del kernel Linux 2.2, i privilegi del superuser sono suddivisi in *capabilities*, abilitabili e disabilitabili indipendentemente[2].

CAP_SETPCAP	Consente la modifica delle capabilities di un processo
CAP_SYS_MODULE	Consente l'inserimento/la rimozione di moduli kernel
CAP_SYS_RAWIO	Consente la modifica della memoria kernel
CAP_SYS_PACCT	Consente la configurazione dell'accounting dei processi
CAP_SYS_NICE	Consente la modifica della priorità di un processo
CAP_SYS_RESOURCE	Consente la sovrascrittura dei limiti delle risorse
CAP_SYS_TIME	Consente la modifica dell'orologio di sistema
CAP_SYS_TTY_CONFIG	Consente la modifica dei dispositivi *tty* (terminali)
CAP_AUDIT_WRITE	Consente la scrittura di audit log
CAP_AUDIT_CONTROL	Consente la configurazione dell'*Audit Subsystem*
CAP_MAC_OVERRIDE	Consente di ignorare la *Kernel MAC Policy*
CAP_MAC_ADMIN	Consente la configurazione MAC
CAP_SYSLOG	Consente la modifica del fuzionamento di *printk*
CAP_NET_ADMIN	Consente la configurazione di rete
CAP_SYS_ADMIN	Attribuisce tutti i privilegi di amministratore

Tabella 2.1: Capabilities disabilitate nei container Docker unprivileged[2]

Essendo eseguiti a livello del sistema operativo, i container non necessitano di tutti i permessi di amministrazione: la rimozione di alcuni privilegi non impatta sull'esecuzione del container nè diminuisce le sue funzionalità, al contrario aumenta il livello di sicurezza nel sistema[2]. Per questo motivo, Docker disabilita di default gran parte delle capabilities messe a disposizione da Linux in modo da prevenire che un'eventuale intrusione possa danneggiare l'ambiente esterno al container compromesso.

SELinux

Il sistema introdotto da SELinux prevede un miglioramento dal punto di vista della sicurezza di sistema: viene introdotto un livello aggiuntivo di controlli sui permessi per l'accesso agli oggetti, chiamato *Mandatory Access Control* (MAC). Con il ter-

mine *oggetti* si fa riferimento a qualsiasi file/cartella/processo presente nel sistema in uso, introducendo un sistema di regole chiamate *policies* divise in 3 categorie[2]:

- *Type Enforcement*, che impone meccanismi di sicurezza basati su tipi/etichette assegnati agli oggetti
- *Multi-Level Security (MLS) Enforcement*, che suddivide gli accessi basandosi su livelli di sicurezza degli oggetti
- *Multi-Category Security (MCS) Enforcement*, che suddivide gli accessi basandosi sulle categorie di appartenenza degli oggetti

Attraverso il miglioramento della sicurezza offerto da SELinux il kernel viene incaricato del controllo degli accessi sugli oggetti, garantendo una prevenzione che processi interni ad un container (anche con privilegi di root) possano interagire con oggetti esterni ad esso[2]. Docker utilizza due delle politiche di sicurezza offerte da SELinux elencate in precedenza: Type Enforcement, con lo scopo di proteggere il sistema host dai processi in esecuzione all'interno dei container, e MLS enforcement, per prevenire attacchi container-container.

AppArmor

L'utilizzo di AppArmor pone un ulteriore livello di sicurezza basato sul *Mandatory Access Control*, come quanto visto con SELinux, restringendo il suo utilizzo ai singoli programmi[2]. L'amministratore di sistema viene abilitato all'inserimento di un profilo di sicurezza per ogni programma, in grado di limitarne le *capabilities*. Esistono due modalità supportate da AppArmor:

- *enforcement mode*, per applicare le politiche definite nel profilo in modo da bloccare e loggare i tentativi di violazione
- *complain/learning mode*, per permettere le violazioni delle politiche definite, garantendo un'operazione di log utile allo sviluppo di nuovi profili di sicurezza

Docker viene integrato con AppArmor attraverso un'interfaccia per il caricamento di profili predefiniti all'avvio di un nuovo container[2]: il profilo viene caricato in *enforcement mode*, limitando i processi interni al container e bloccando gli eventi identificati come violazioni.

2.2 Modelli di minaccia

Nonostante la presenza delle tecnologie elencate in precedenza, Docker risulta vulnerabile a vari tipi di attacchi. I modelli di minaccia trattati a seguito risultano essere alcuni tra i più diffusi attacchi informatici nei confronti dei sistemi di virtualizzazione basati su container (e non solo).

Kernel Exploit

Il kernel è l'elemento del sistema operativo incaricato della gestione di tutte le operazioni legate all'hardware, i processi in esecuzione e, in questo caso, i container. Nel caso di un *exploit* del kernel, tutte le applicazioni in esecuzione all'interno dei container sono a rischio compromissione[18].

Nel caso in cui un'applicazione all'interno di un container risultasse compromessa e ottenesse dei diritti privilegiati sul kernel del sistema, l'attaccante avrebbe la possibilità di compromettere tutti gli altri container in esecuzione nel sistema host sotto attacco: questo particolare rischio è dovuto all'utilizzo della stessa architettura kernel da parte di tutti gli ambienti virtuali in esecuzione attraverso container[18].

Possibili contromisure

Delle possibili contromisure per questa tipologia di attacco, raccomandate direttamente da Docker, riguardano l'utilizzo di almeno una feature di sicurezza tra AppArmor o SELinux, ilustrate in precedenza. L'adozione di almeno uno tra questi sistemi garantirebbe un ambiente cloud più sicuro dal punto di vista dei kernel exploit.

Altri metodi utili alla protezione riguardano[18]:

- la mappatura in gruppi dei container mutualmente fidati
- l'esecuzione di applicazioni non fidate con privilegi esclusivamente non-root
- l'utilizzo dei *namespaces*
- l'impostazione in sola lettura del filesystem dei container
- il blocco dell'*inter-container communication*
- l'installazione dei soli pacchetti ritenuti essenziali

Attacchi DoS

Come anticipato in precedenza, gli attacchi DoS (Denial-of-Service) sono uno tra gli attacchi più diffusi soprattutto per quanto riguarda le risorse di rete: questa tipologia di attacchi vede la presenza di uno o più processi con lo scopo di consumare le risorse di un sistema, portandolo all'impossibilità di fornire un servizio nei confronti degli altri host (o processi) che lo richiedono[18].

Nel sistema implementato da Docker tutti i container condividono le risorse del kernel: una conseguenza derivante da questa implementazione è che, in caso di attacco DoS da parte di un container interno al sistema, tutti gli altri container risulterebbero privati dell'accesso alla risorsa richiesta.

Possibili contromisure

Tra le contromisure di questo tipo di attacco, rientrano alcune delle feature elencate in precedenza:

- l'isolamento dei processi
- l'isolamento del filesystem
- l'isolamento dei dispositivi
- l'isolamento IPC
- l'isolamento di rete
- la limitazione delle risorse

Grazie al controllo sull'allocazione delle risorse destinate a ciascun container può avvenire una prevenzione riguardo questa tipologia di attacchi. Questa limitazione può avvenire attraverso il controllo e la gestione tramite i *cgroups*, trattati in precedenza, in grado di limitare l'accesso risorse critiche come la memoria centrale, le periferiche, le memorie di massa e il tempo di accesso alla CPU: l'adozione di questo strumento assicura che un container compromesso possa monopolizzare delle risorse destinate all'uso comune all'interno di un host[18].

Container Breakout

In questa tipologia di attacchi, l'attaccante si pone nella condizione di sfuggire dalle limitazioni imposte dal container compromesso, avendo di fatto accesso al sistema host e agli altri container in esecuzione.

Una delle metodologie di attacco riguarda l'utilizzo della funzione `open_by_handle_at()`, che permette l'accesso ad un filesystem montato attraverso la struttura *file_handle*; questa funzione, per poter essere utilizzata, necessita della capability CAP_DAC_READ_SEARCH, garantita di default ad un utente con permessi di superuser all'interno di un container. In questo modo l'attaccante può superare i sistemi di sicurezza imposti e accedere a file esterni all'ambiente virtuale[18].

Possibili contromisure

Secondo quanto riportato dal sito di Docker, questa vulnerabilità risulta presente fino all'adozione di Docker 1.0 (ex Docker 0.12). Ulteriori contromisure riguardano

- l'impostazione del filesystem del container in modalità di sola lettura
- l'esecuzione dei container in modalità non privileged

L'esecuzione di container in modalità privileged, come detto nei capitoli precedenti, garantisce tutti i permessi e le capabilities del sistema host. Ne risulta quindi un grande rischio dal punto di vista della sicurezza in caso di attacchi interni a seguito della compromissione di un container.

Poisoned Images

Come riportato in precedenza, le immagini costituiscono la base per la creazione e l'esecuzione di container Docker: possono essere ricevute da una libreria pubblica, come Docker Hub, o create liberamente e condivise. Per immagine "poisoned" si intende la presenza di software malevolo all'interno di un'immagine apparentemente sicura.

Un'immagine, a seguito del download, viene verificata basandosi sulla presenza di un *manifest* firmato digitalmente. A seguito del controllo della firma, tuttavia, Docker non prevede un meccanismo di autenticazione del checksum dell'immagine presente del manifest[18]: ciò non previene la possibilità che un'immagine "poisoned" possa venire trasmessa insieme ad un manifest firmato e verificato.

La pipeline messa in atto da Docker per il download delle immagini è pratica ma estremamente insicura per quel che riguarda l'autenticazione[18]:

1. *decompressione* del file scaricato
2. *tarsum*, finalizzato all'ottimizzazione delle prestazioni
3. *unpack*, rivolto alla decodifica di quanto scaricato e il salvataggio (di file non verificati) in memoria di massa

Possibili contromisure

Le contromisure proposte riguardano principalmente il download delle sole immagini fidate e autenticate e l'autenticazione manuale delle immagini scaricate prima di essere importate all'interno di Docker ed eseguite.

Attacchi MITM

Gli attacchi MITM (Man-in-the-Middle) costituiscono una grande minaccia per la sicurezza delle comunicazioni. Questa tipologia di attacchi prevede che, durante una comunicazione tra due host, avvenga l'inserimento di un terzo utente malintenzionato con lo scopo di monitorare la comunicazione, alterarla oppure rubare informazioni riservate.

Possibili contromisure

Una delle contromisure più efficaci nei confronti degli attacchi MITM riguarda l'isolamento di rete, trattato in precedenza: è necessario che i container siano configurati in modo tale da non consentire la manipolazione dei dati o l'intercettazione di una comunicazione in corso[18].

OpenVPN (un software open source per la creazioni di reti private virtuali) implementa una valida contromisura agli attacchi MITM, permettendo la creazione di VPN che utilizzano un sistema di crittografia TLS per i messaggi in transito[18]. Docker prevede l'integrazione di un server OpenVPN preimpostato in modo da essere eseguito con lo scopo di mantenere una comunicazione sicura tra i container e il sistema host.

ARP Spoofing

Il protocollo ARP (Address Resolution Procotol) garantisce la conversione tra indirizzi IP (livello rete) e indirizzi MAC (livello data-link) all'interno di una rete locale. Gli attacchi di tipo ARP spoofing sono costituiti dalla falsificazione di messaggi legati al protocollo ARP e al loro invio all'interno di una rete LAN (Local Area Network), con lo scopo di associare l'indirizzo MAC dell'attaccante con l'indirizzo IP della vittima, mettendo l'utente malevolo nella condizione di intercettare il traffico destinato all'indirizzo IP indicato.

Questo tipo di attacco, applicato ad un sistema come quello implementato da Docker, consentirebbe ad un container di intercettare tutto il traffico diretto alla scheda

di rete del sistema host o ad altri container: ciò si verifica a causa dell'assenza di filtri e meccanismi di sicurezza nel protocollo ARP[18]. A causa di questa vulnerabilità un attaccante potrebbe essere in grado, dopo aver compromesso un container, di intercettare pacchetti contenenti informazioni sensibili scambiate tra i vari container all'interno del sistema host (ad esempio le credenziali scambiate tra un'applicazione web e il suo database) e di inserire dei payload malevoli nelle comunicazioni di rete[18].

Possibili contromisure

Una delle contromisure più potenti riguarda la possibilità di eseguire i container rimuovendo loro la capability NET_RAW: in questo modo i processi interni al container non saranno in grado di creare socket di tipo PF_PACKET, senza i quali un attacco ARP non può essere eseguito[18].

Un altro metodo riguarda l'utilizzo di *ebtables* per filtrare i frame Ethernet, garantendo la rilevazione e successiva rimozione dei pacchetti ARP con informazioni errate o sospette[18].

Capitolo 3

Attacchi a Docker

In questo capitolo verranno presentate le principali *CVE* (*Common Vulnerabilities and Exposures*) inerenti all'ecosistema Docker, passando successivamente a delle tipologie di attacchi in grado di sfruttarle per compromettere un container o il sistema host in uso. Successivamente verrà analizzata la presenza di vulnerabilità nel principale archivio pubblico di immagini, Docker Hub.

3.1 CVE note

Attraverso il termine *CVE* viene indicato un insieme di vulnerabilità e falle relative alla sicurezza di software e sistemi informatici, presentato come una lista pubblicamente accessibile e che prevede l'assegnazione di un ID univoco ad ogni vulnerabilità riscontrata[12].

L'identificazione di nuove vulnerabilità inizia molto frequentemente con la segnalazione da parte di organizzazioni e membri della comunità open source, e solo successivamente a una fase di verifica è previsto l'inserimento in un database pubblico contenente una breve descrizione e riferimenti.

Esistono degli specifici criteri perché una CVE possa essere considerata tale[12]:

1. la vulnerabilità deve poter essere *corretta indipendentemente* da altri bug o CVE presenti
2. deve essere presente una *conferma da parte del produttore* del sistema vulnerabile oppure deve essere dimostrata la *violazione delle politiche di sicurezza*

3. la vulnerabilità deve essere *riconducibile ad un singolo prodotto*, e quindi univoca

3.1.1 CVE relative a Docker

Nonostante le varie contromisure messe in atto, Docker non è esente dalla presenza di vulnerabilità e rischi per la sicurezza dei sistemi che lo utilizzano. Spesso le CVE riscontrate sono riconducibili alle librerie utilizzate per l'implementazione del sistema stesso.

In questa sezione verranno analizzate alcune vulnerabilità riscontrate nell'ecosistema Docker, con un particolare focus alle CVE che potrebbero essere sfruttate per mettere in atto alcuni degli attacchi elencati in precedenza.

CVE-2020-15257

Containerd è un ambiente di esecuzione per la gestione di container all'interno di un sistema, con lo scopo di gestire creazioni, avvii, fermate e rimozioni degli ambienti virtuali attraverso un'interfaccia API.

Questo sistema non risulta esente da vulnerabilità in grado di porre sotto attacco un host o un ambiente virtuale: CVE-2020-15257 descrive un mancato controllo sulla restrizione dell'accesso ad un socket UNIX astratto. L'assenza del controllo potrebbe permettere ad un container compromesso di inviare con successo richieste API relative alla creazione di nuovi ambienti virtuali, eseguiti come processi con privilegi elevati, estendendo le possibilità di attacco[11]. Gli aggiornamenti alle versioni di containerd 1.3.9 e 1.4.3 hanno previsto la risoluzione di questa problematica implementando un controllo sugli accessi al socket UNIX in questione.

CVE-2019-16884

Grazie ad un bug presente in runC (fino alla versione 1.0.0-rc8) è possibile aggirare le limitazioni imposte da AppArmor a causa di alcuni controlli non corretti effettuati su dispositivi montati.

La vulnerabilità in questione rende possibile montare `/proc` all'interno di un container e aggirare le misure di sicurezza in quanto il profilo AppArmor viene definito all'interno di `/proc/self/attr/apparmor/current`[17].

CVE-2019-5736

Una delle vulnerabilità più serie e recenti riguardanti l'ecosistema Docker è CVE-2019-5736, riguardante runC fino alla versione 1.0-rc6 (utilizzato fino alla versione di Docker 18.09.2).

Questa vulnerabilità può portare alla capacità di sovrascrivere runC, permettendo all'attaccante di ottenere privilegi di root all'interno del sistema attraverso la possibilità di eseguire un comando come utente privilegiato all'interno di un nuovo container creato a partire da un'immagine controllata dall'attaccante o in un container esistente in cui l'attaccante ha accesso in scrittura[4].

Quando un container docker viene creato o quando il comando `docker exec` viene utilizzato, viene eseguito un processo runC con lo scopo di avviare il container. A causa della presenza di un link simbolico tra `/proc/self/exe` e il binario di runC nel sistema host è possibile richiedere a runC di eseguire sè stesso all'interno di un container: l'utente root interno al container è quindi in grado di modificare il file binario di runC nel sistema host attraverso questo collegamento[17].

Alla successiva esecuzione di runC (o di `docker exec`) verrà eseguito il binario sovrascritto, rendendo possibile l'esecuzione di codice nel sistema host da parte di un container compromesso.

CVE–2019–5021

Una vulnerabilità relativa ad una configurazione insufficiente riguarda l'immagine Docker di Alpine Linux nelle versioni 3.3, 3.4 e 3.5. Nello specifico, la password dell'utente `root` interno al container viene lasciata vuota: grazie a questo problematica è possibile ottenere privilegi di root internamente al container fornendo una stringa vuota[17].

A questa vulnerabilità sono stati attribuiti dei punteggi *CVSS 9.8/10* e *CVSS2 10/10*, classificandola come una vulnerabilità ad altissimo rischio senza considerare il contesto in cui può essere applicata. Un punteggio così alto è dovuto alla possibilità di ottenere i privilegi di root fornendo una stringa vuota ma il fatto di essere applicabile solamente all'interno di un ambiente isolato (in questo caso un container) la rende di minore rilevanza. Nel caso relativo all'ecosistema Docker un attaccante è in grado di eseguire codice con privilegi di root internamente al container ma dovrebbe comunque trovare un'ulteriore vulnerabilità per riuscire ad effettuare un *container breakout*: CVE–2019–5021 di fatto può mettere in una situazione di vantaggio l'attaccante, ma non garantisce la possibilità di attaccare con successo il sistema host.

Codice 3.1: Esempio relativo a CVE–2019–5021

```
(host) docker run -it --rm alpine:3.5 cat /etc/shadow
root:::0:::::
...
(host) docker run -it --rm alpine:3.5 sh
(container) apk add --no-cache linux-pam shadow
...
(container) adduser test
...
(container) su test
Password:
(container) su root
(container) whoami
root
```

Nell'esempio presentato[17] viene inizialmente dimostrata l'assenza della password per l'utente root in un container Alpine 3.5. A seguito viene effettuata una connessione tramite shell in modalità interattiva (flag `-it`), vengono aggiunti dei pacchetti per la gestione di password e autorizzazioni e viene creato un nuovo utente `test`: da questo utente test è possibile ottenere nuovamente i privilegi di root senza inserire alcuna password tramite il comando `su root`.

3.2 Container breakout

Come riportato in precedenza, gli attacchi di tipo container breakout sono tra i più rischiosi dal punto di vista della sicurezza per un sistema come Docker, che fa di uno dei suoi punti di forza l'isolamento dei sistemi virtuali dall'host in uso.

In questa sezione verrà analizzato il rischio posto da questa tipologia di attacchi e alcuni possibili metodi per prevenire la possibilità che un container compromesso possa avere accesso ed eseguire codice sul sistema host.

È possibile raggruppare le vulnerabilità che portano ad un container breakout attraverso le seguenti categorie[13]:

- derivanti da una *cattiva gestione di descrittori dei file*
- eseguiti a seguito di un *mancato controllo dell'accesso a runtime*
- derivanti da un'*esecuzione da parte del sistema host* di oggetti condivisi

3.2.1 Cattiva gestione dei file descriptor

All'interno di un sistema operativo Unix, un *file descriptor* viene utilizzato per rappresentare un oggetto aperto da un processo su cui possono essere effettuate operazioni di lettura e scrittura.

Questa tipologia di problema si presenta nel momento in cui un descrittore del file risulta accessibile internamente al container attraverso il filesystem `/proc`, dando accesso in lettura e scrittura al sistema host[13]. Come riportato in precedenza per CVE-2019-5736 (3.1.1), è possibile creare un riferimento al runtime del container in uso riuscendo ad ottenere

1. un link simbolico a partire da `/bin/bash` (interno al container) verso `/proc/self/exe` (nel sistema host)
2. un file per la creazione di una backdoor

Nel momento dell'esecuzione del container viene creato un collegamento tra il file binario appartenente a runC e `/proc/self/exe`, portando il sistema host ad eseguire il contesto del container e caricare il file malevolo citato in precedenza. L'esecuzione di questo file porta alla creazione di una backdoor nel sistema host, portando a termine l'attacco di breakout.

Simulazione di attacco

I requisiti per la simulazione di attacco prevedono:

- una versione di Docker precedente a 18.09.2 nel sistema target
- l'accesso ai file presenti in https://github.com/rancorzinho/pocs contenenti il payload
- l'installazione di *netcat* (software per la comunicazione remota a riga di comando) sul sistema attaccante

Codice 3.2: Simulazione di attacco per CVE–2019–5736

```
(attaccante) nc -nlvvp 4455
Listening on [any] 4455 ...

(host) docker run -it --privileged ubuntu --name cve /bin/bash
(container) git clone https://github.com/rancorzinho/pocs.git
...
(container) cd pocs/CVE-2019-5736
(container) make
```

```
(container) ./pwn.sh

(host) docker exec -it cve /bin/sh

(attaccante)
connect to [172.17.0.1] from (UNKNOWN) [172.20.86.130] 50784
sh: 0: cannot access tty; job control turned off
(attaccante) id
uid=0(root) gid=0(root) groups=0(root)
```

Il codice presentato[10] permette di portare a termine un attacco di tipo container breakout verso un sistema che presenta una versione di Docker precedente a 18.09.2.

L'operazione iniziale prevede l'apertura di una *reverse shell* da parte della macchina attaccante tramite netcat, in ascolto su una porta a scelta (in questo caso la 4455). Successivamente è possibile procedere all'esecuzione di un container Docker all'interno del sistema target, nel quale eseguire il download del payload riportato in precedenza e dove impostare l'indirizzo IP dell'attaccante nel campo HOST all'interno del file `pocs/CVE-2019-5736/payload.c`.

Dopo l'esecuzione del comando `make` e di `./pwn.sh` all'interno del container sarà possibile attendere la successiva operazione di `exec` da parte del sistema host da attaccare, a seguito della quale verrà aperta una comunicazione tra netcat nel sistema attaccante e il sistema target, dando così la possibilità di eseguire comandi e installare una backdoor.

3.2.2 Mancato controllo dell'accesso a runtime

Attraverso un controllo assente assente sugli accessi, l'attaccante può ottenere il controllo completo su un componente runtime del sistema host: un esempio ricade nella vulnerabilità CVE-2020-15257 (3.1.1), la quale a causa di un mancato controllo permette l'accesso ad un socket UNIX astratto fornito da *containerd*[13]. Attraverso questo canale di comunicazione è possibile creare o avviare nuovi container all'interno del namespace del sistema host senza alcun controllo da parte di sistemi citati in precedenza come AppArmor o seccomp e con tutte le capabilities abilitate, creando di fatto un container *privileged* attraverso il quale è possibile avere accesso al sistema host.

3.2.3 Esecuzione su oggetti condivisi

Un ulteriore metodo di attacco per l'evasione dall'isolamento predisposto da un container riguarda l'esecuzione interna al contesto del container da parte di un binario del sistema host: l'attaccante è in grado di manipolare l'esecuzione attraverso un oggetto condiviso o un link simbolico[13].

Un esempio è CVE-2019-14271, la quale pone un problema relativo alla sicurezza per l'implementazione del comando Docker `cp` il quale permette di copiare file da, verso e tra container. Questa vulnerabilità è stata risolta con l'aggiornamento alla versione di Docker 19.03.1.

Le versioni vulnerabili di Docker risultano compilate con Go v1.11, versione nella quale alcuni pacchetti contenenti del codice C prevedevano il caricamento di librerie condivise a runtime: nel caso del comando `cp`, il quale si avvale di un processo helper chiamato `docker-tar`, le librerie vengono caricate dal filesystem del container e non da quello del sistema host, causando l'esecuzione di codice presente all'interno del container nel namespace dell'host[1]. Questa esecuzione viene effettuata con tutte le capabilities attive, senza limitazioni da parte di cgroups o seccomp, portando così il container compromesso ad avere un accesso *root* al sistema host.

Simulazione di attacco

Per sfruttare CVE-2019-14271 è necessaria la creazione una libreria *libnss* (orientata a Network Security Services) malevola, come riportato dal lavoro di Y. Avrahami[1]: l'autore predispone la modifica della libreria originale `libnss_files.so` aggiungendo la funzione `run_at_link()` in modo che venga eseguita come funzione di inizializzazione per la libreria stessa nel momento in cui viene caricata dal processo (in questo caso `docker-tar`).

La funzione in questione controlla inizialmente se l'esecuzione sia all'interno del contesto di `docker-tar`, successivamente sostituisce la libreria modificata con quella originale per evitare che successivi caricamenti della stessa possano far ripartire l'esecuzione della funzione `run_at_link()`. Per semplificare l'exploit e consentirne la scrittura in un linguaggio diverso dal C viene tentata l'esecuzione di un file in una cartella interna al container indicata dall'attaccante, rendendo la logica per il breakout indipendente da quella per l'ottenimento delle capabilities massime attraverso la CVE in questione.

Lo script bash proposto per il breakout consiste nel mount del filesystem del sistema host all'interno di `/host_fs` nel container, rendendo così accessibile il sistema esterno e aggirando il meccanismo di isolamento del filesystem introdotto da Docker (2.1.1).

3.2.4 Contromisure

Come evidenziato in precedenza, le CVE sfruttabili da parte di un attaccante per compiere un container breakout riguardano solamente alcune versioni di Docker e degli altri software presentati. Per questo motivo è di estrema importanza mantenere un sistema il più aggiornato possibile in modo da evitare la presenza di vulnerabilità conosciute: nel momento di pubblicazione di una CVE i team di sviluppo provvedono a rilasciare un nuovo aggiornamento per fornire una *patch* il prima possibile, in modo da limitare attacchi su larga scala (soprattutto nei casi di vulnerabilità classificate come "gravi", come alcune di quelle elencate in precedenza).

Come descritto in precedenza (2.1) le feature di sicurezza messe in atto da Docker prevedono un uso dei *namespaces* forniti da Linux per creare vari livelli di isolamento tra container e host, rendendo possibile la differenziazione tra utenti interni ed esterni[13].

Per default un container viene eseguito con i privilegi dello stesso utente che lo inizializza: secondo lo studio riportato da M. Reeves et al. [13] il 58% dei container viene eseguito come utente root, ereditandone i privilegi. Grazie all'utilizzo di meccanismi come quello relativo ai *namespaces* sarebbe possibile fermare gran parte dei tentativi di exploit delle CVE elencate in precedenza, andando di fatto a separare l'utente interno al container da quello presente nel sistema host.

3.3 Pull di immagini non sicure

Come già anticipato, uno dei modelli di minaccia principali per l'ambiente Docker riguarda le cosiddette *poisoned images*(2.2), le quali consistono in un mancato meccanismo di autenticazione del *checksum* inserito nel manifest, consentendo il download di immagini compromesse con allegato un manifest firmato digitalmente. Un'ulteriore problematica riguarda la presenza di immagini su Docker Hub che presentano vulnerabilità con alte priorità relative sulla sicurezza, ponendo gli utilizzatori in una situazione di pericolo.

3.3.1 Poisoned images

La fase di download delle immagini provenienti da Docker Hub prevede un sistema di verifica basato su una firma digitale del file manifest (contenente il checksum di quanto si sta scaricando) allegato all'immagine stessa, senza però verificare che il checksum contenuto corrisponda all'oggetto del download. In questo modo si

apre la possibilità per un attaccante di inviare una qualsiasi immagine malevola con allegato un manifest contenente una firma verificata, facendo contrassegnare l'intero download come "verificato" e "sicuro"[15].

Come riportato in precedenza, la pipeline predisposta da Docker per il download delle immagini prevede come passaggi la *decompressione* del file scaricato, il *tarsum* e l'*unpack*. Queste 3 fasi presentano varie insicurezze e vulnerabilità:

1. la fase di **decompressione** supporta 3 algoritmi. `gzip` e `bzip2` sono considerati *memory-safe*, e quindi vulnerabili principalmente ad attacchi di tipo DoS mirati a sovraccaricare la CPU o l'utilizzo della memoria; il terzo algoritmo è `xz`, il quale non è *memory-safe* e nel caso di un input malevolo sarebbe possibile l'esecuzione di codice. In Docker questa situazione è peggiorata dall'esecuzione di `xz` con privilegi di root.
2. la fase di **tarsum** prevede la decodifica del tar ottenuto e l'hashing delle sue porzioni con un ordine deterministico. Le vulnerabilità presenti in questa fase riguardano attacchi di tipo DoS o vulnerabilità logiche in grado di causare modifiche al flusso di lavoro o al file stesso.
3. la fase di **unpack** consiste nella decodifica del file tar e il salvataggio dei suoi file su disco: questo tipo di approccio può risultare estremamente pericoloso per l'intero sistema in quanto è previsto il salvataggio diretto di file non verificati

Contromisure

Alcune delle contromisure più effettive riguardano l'abbandono del sistema *tarsum* e la verifica effettiva dell'immagine scaricata prima di procedere a qualsiasi altra azione che la possa coinvolgere. Può risultare utile l'aggiunta di un livello di isolamento per quel che riguarda le fasi di decompressione e unpack, in modo che vengano eseguite con i privilegi minimi[15].

3.3.2 Docker Hub

Come già anticipato (1.1.1), Docker Hub risulta essere ad oggi il registro pubblico di immagini a maggiore utilizzo: attraverso di esso è possibile effettuare operazioni di *pull* di immagini prima di poterle eseguire sotto forma di container: per tale motivo è molto importante che le immagini archiviate e pubblicamente disponibili non presentino vulnerabilità in grado di compromettere il sistema host degli utilizzatori.

Dallo studio di T. Desikan ([5]) è emerso che più del 30% delle immagini ufficiali pubblicamente disponibili risultano vulnerabili ad attacchi segnalati come "ad alto rischio", in grado di minacciare il sistema in uso.

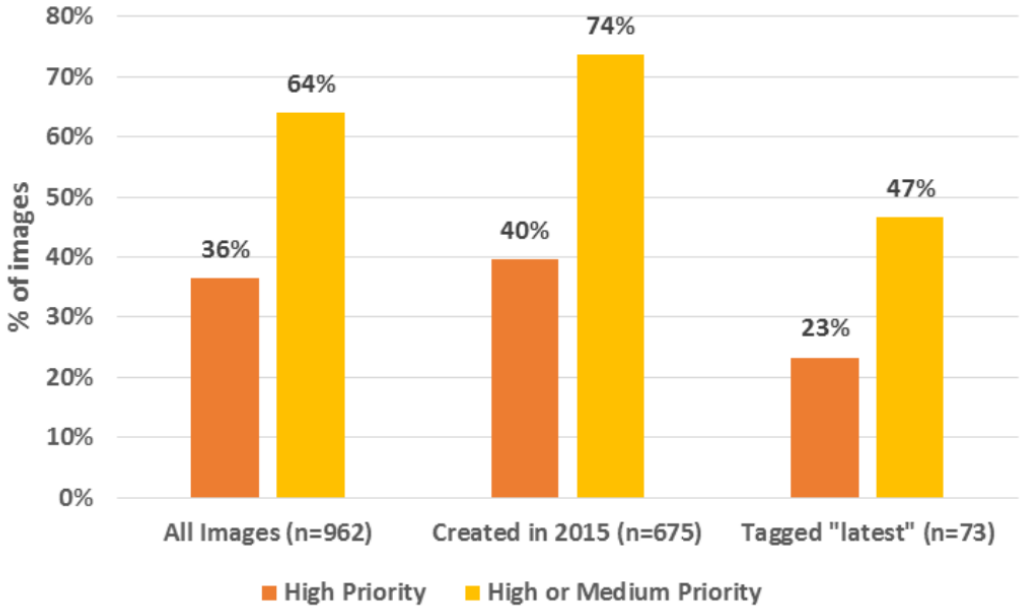

Figura 3.1: Percentuale di immagini con vulnerabilità su Docker Hub[5]

Dalla figura precedente è possibile notare l'alta presenza di immagini che presentano vulnerabilità ad alta o media priorità, situazione lievemente migliorata ma non ottimale tra le immagini con tag "latest". In questi numeri sono alcuse alcune tra le immagini più utilizzate da parte degli utenti in fase di pull e usate come base per ulteriori progetti, propagando le vulnerabilità presenti.

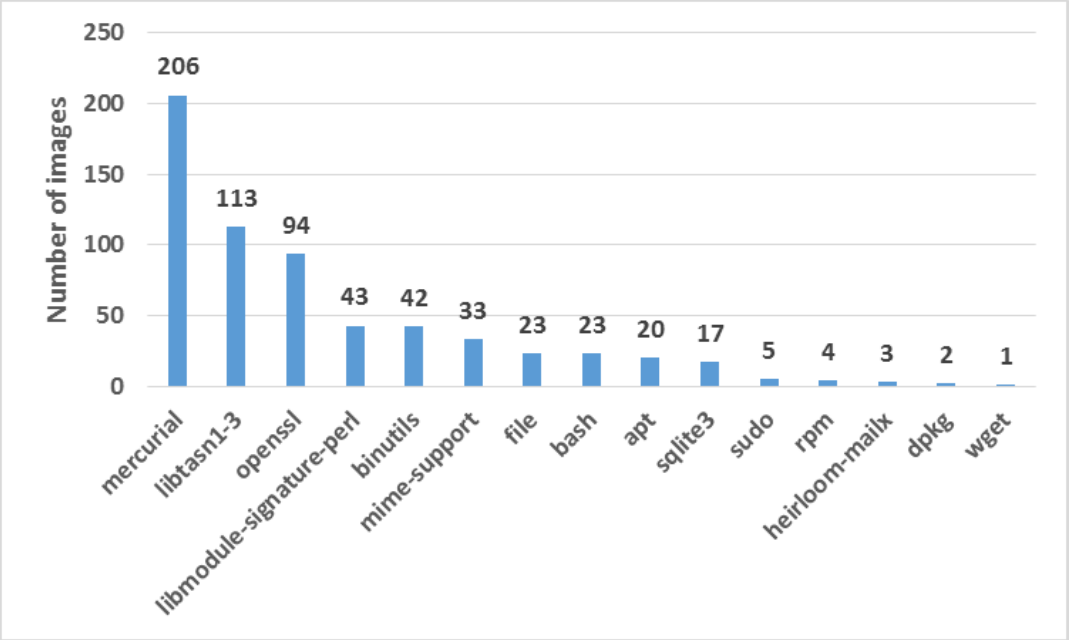

Figura 3.2: Fonti delle vulnerabilità rilevate su Docker Hub[5]

La figura indica in maniera più approfondita l'origine delle vulnerabilità rilevate, andando ad individuare *mercurial* (un software per il controllo di versione del codice sorgente) con CVE-2014-9462 come vulnerabilità più diffusa. Altre vulnerabilità presenti degne di nota riguardano *Heartbleed* e *Poodle* per *openssl* e *ShellShock* per *bash*.

Come riportato in precedenza, l'analisi condotta riguarda immagini ufficiali presenti in Docker Hub. Sarebbe inoltre possibile analizzare le vulnerabilità presenti nelle centinaia di migliaia di immagini fornite dagli utenti: uno studio simile è stato condotto a seguito dell'analisi iniziale da parte di T. Desikan ([5]), andando a rilevare un campione statistico di 1700 immagini. Le percentuali di vulnerabilità risultano sensibilmente più alte rispetto allo studio iniziale basato su immagini ufficiali, delineando una mancanza di controlli preventivi alla pubblicazione su Docker Hub. Le vulnerabilità più comuni in questo caso risultano essere legate a *openssl*, *bash* e *apt*, sintomo di una costruzione a partire dalle immagini Ubuntu, ereditandone le vulnerabilità.

Capitolo 4

Conclusioni e sviluppi futuri

L'utilizzo dei sistemi di virtualizzazione introduce grandi vantaggi applicativi legati alle capacità di isolamento e di esecuzione in sistemi multi-tenant.

L'adozione di un sistema basato su container, come Docker, accentua questi vantaggi attraverso una maggiore portabilità e un minore peso sul sistema in uso grazie alla sua capacità di interfacciarsi direttamente con il kernel host; questo, però, risulta uno dei principali punti deboli relativi alla sicurezza, rendendo necessari vari livelli di isolamento e maggiori controlli sulle operazioni effettuate. La metodologia alla base della sicurezza di questi sistemi è in continua evoluzione, riuscendo però a fornire attualmente un buon livello di protezione.

Secondo quanto riportato attraverso il lavoro svolto, infatti, i principali pericoli riguardano un numero ristretto di vulnerabilità specifiche che risultano risolte nelle versioni più aggiornate dei sistemi adottati: per questo motivo risulta evidente la necessità di mantenere costantemente aggiornato il sistema usato, non solo dal punto di vista delle versioni dei software per la virtualizzazione ma anche delle immagini stesse.

Per quanto riguarda un possibile sviluppo futuro, potrebbe risultare interessante analizzare in maniera automatizzata e più approfondita le vulnerabilità presenti nelle immagini attualmente disponibili su Docker Hub, espandendo uno degli ultimi argomenti affrontati nel corso di questo lavoro.

Bibliografia

[1] Y. Avrahami. *Docker Patched the Most Severe Copy Vulnerability to Date With CVE-2019-14271*. Last visit: 20/11/2023. URL: https://unit42.paloaltonetworks.com/docker-patched-the-most-severe-copy-vulnerability-to-date-with-cve-2019-14271/.

[2] T. Bui. "Analysis of Docker Security". In: *CoRR* abs/1501.02967 (2015). arXiv: 1501.02967. URL: http://arxiv.org/abs/1501.02967.

[3] T. Combe, A. Martin e R. Di Pietro. "To Docker or Not to Docker: A Security Perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62. DOI: 10.1109/MCC.2016.100.

[4] CVEdetails. *CVE-2019-5736*. Last visit: 06/11/2023. URL: https://www.cvedetails.com/cve/CVE-2019-5736/.

[5] T. Desikan. *Over 30Priority Security Vulnerabilities*. Last visit: 22/11/2023. 5 Mag. 2015. URL: https://www.banyansecurity.io/blog/over-30-of-official-images-in-docker-hub-contain-high-priority-security-vulnerabilities/.

[6] Docker. *Docker overview*. Last visit: 26/09/2023. URL: https://docs.docker.com/get-started/overview/.

[7] Docker. *Introducing runC: a lightweight universal container runtime*. Last visit: 10/11/2023. URL: https://www.docker.com/blog/runc/.

[8] Docker. *What is a Container?* Last visit: 29/09/2023. URL: https://www.docker.com/resources/what-container/.

[9] Linux. *capabilities(7) — Linux manual page*. Last visit: 19/10/2023. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html.

[10] Rancorzinho. *pocs*. https://github.com/rancorzinho/pocs. 2019.

[11] RedHat. *CVE-2020-15257*. Last visit: 16/11/2023. URL: https://access.redhat.com/security/cve/cve-2020-15257.

[12] RedHat. *What is a CVE?* Last visit: 06/11/2023. URL: https://www.redhat.com/en/topics/security/what-is-cve.

[13] M. Reeves, D. J. Tian, A. Bianchi e Z. B. Celik. "Towards Improving Container Security by Preventing Runtime Escapes". In: *2021 IEEE Secure Development Conference (SecDev)*. 2021, pp. 38–46. DOI: 10.1109/SecDev51306.2021.00022.

[14] E. Reshetova, J. Karhunen, T. Nyman e N. Asokan. "Security of OS-level virtualization technologies: Technical report". In: *CoRR* abs/1407.4245 (2014). arXiv: 1407.4245. URL: http://arxiv.org/abs/1407.4245.

[15] J. Rudenberg. *Docker Image Insecurity*. Last visit: 22/11/2023. 23 Dic. 2014. URL: https://titanous.com/posts/docker-insecurity.

[16] VMware. *What is a hypervisor?* Last visit: 27/09/2023. URL: https://www.vmware.com/topics/glossary/content/hypervisor.html.

[17] J. Vrancken. "A Methodology for Penetration Testing Docker Systems". Bachelor thesis. Radboud University, 2020. URL: https://www.cs.ru.nl/bachelors-theses/2020/Joren_Vrancken___4593847___A_Methodology_for_Penetration_Testing_Docker_Systems.pdf.

[18] R. Yasrab. "Mitigating Docker Security Issues". In: *CoRR* abs/1804.05039 (2018). arXiv: 1804.05039. URL: http://arxiv.org/abs/1804.05039.

Elenco delle tabelle

Elenco delle figure

Elenco dei codici

Glossario

ARP spoofing Attacco di rete in cui l'attaccante invia messaggi ARP falsificati in una rete locale

Attacco DoS Attacco di rete in cui l'attaccante sovraccarica le risorse del sistema bersaglio con lo scopo di impedire un servizio

Attacco MITM Attacco di rete in cui l'attaccante si inserisce nella comunicazione tra due host, alterando o intercettando i dati trasmessi

Capability Struttura dati per attribuire permessi consentendo operazioni specializzate

Cgroup Funzionalità del kernel Linux per la limitazione di risorse di una raccolta di processi

Container Ambiente virtuale il cui kernel è condiviso con il sistema host

Exploit Procedura attraverso la quale il sistema sotto attacco compie operazioni non previste dal normale comportamento

File descriptor Rappresentazione di un oggetto aperto da un processo in un sistema operativo Unix su cui possono essere effettuate operazioni di lettura e scrittura

Kernel Componente centrale di un sistema operativo con lo scopo di fare da tramite tra risorse hardware e processi software in esecuzione

Multi-tenant Abilità di un sistema di servire più clienti in modo isolato e sicuro sulla stessa infrastruttura

Namespace Funzionalità del kernel Linux per l'isolamento di risorse e la creazione di ambienti isolati

Reverse shell Tecnica per eseguire comandi remoti sulla macchina vittima, la quale richiede l'apertura della connessione

Virtualizzazione Partizionamento di un sistema in ambienti virtuali multipli ed isolati

Acronimi

ARP Address Resolution Protocol

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

DoS Denial-of-Service

IPC Inter-Process Communication

LAN Local Area Network

PID Process ID

SELinux Security Enhanced Linux

TLS Transport Layer Security

VPN Virtual Private Network